Problem Set 7 Solutions

This problem set is due **in recitation** on **Friday, May 7**.

Reading: Chapter 22, 24, 25.1-25.2, Chapters 34, 35

There are **five** problems. Each problem is to be done on a **separate sheet** (or sheets) of paper. Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated. As on previous assignments, "give an algorithm" entails providing a description, proof, and runtime analysis.

Problem 7-1. Arbitrage

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. Suppose, 1 U.S. dollar bought 0.82 Euro, 1 Euro bought 129.7 Japanese Yen, 1 Japanese Yen bought 12 Turkish Lira, and one Turkish Lira bought 0.0008 U.S. dollars.

Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02$ U.S. dollars, thus turning a 2% profit. Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.

(a) Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that:

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 \; .$$

Solution: We can use the Bellman-Ford algorithm on a suitable weighted, directed graph $G = (V, E)$, which we form as follows. There is one vertex in V for each currency, and for each pair of currencies c_i and c_j, there are directed edges (v_i, v_j) and (v_j, v_i). (Thus, $|V| = n$ and $|E| = \binom{n}{2}$.)
To determine edge weights, we start by observing that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

if and only if

$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} < 1 \; .$$

Taking logs of both sides of the inequality above, we express this condition as

$$\lg \frac{1}{R[i_1, i_2]} + \lg \frac{1}{R[i_2, i_3]} + \lg \frac{1}{R[i_{k-1}, i_k]} + \cdots + \lg \frac{1}{R[i_k, i_1]} < 0 .$$

Therefore, if we define the weight of edge (v_i, v_j) as

$$\begin{aligned} w(v_i, v_j) &= \lg \frac{1}{R[i,j]} \\ &= -\lg R[i,j] , \end{aligned}$$

then we want to find whether there exists a negative-weight cycle in G with these edge weights.

We can determine whether there exists a negative-weight cycle in G by adding an extra vertex v_0 with 0-weight edges (v_0, v_i) for all $v_i \in V$, running BELLMAN-FORD from v_0, and using the boolean result of BELLMAN-FORD (which is TRUE if there are no negative-weight cycles and FALSE if there is a negative-weight cycle) to guide our answer. That is, we invert the boolean result of BELLMAN-FORD.

This method works because adding the new vertex v_0 with 0-weight edges from v_0 to all other vertices cannot introduce any new cycles, yet it ensures that all negative-weight cycles are reachable from v_0.

It takes $\Theta(n^2)$ time to create G, which has $\Theta(n^2)$ edges. Then it takes $O(n^3)$ time to run BELLMAN-FORD. Thus, the total time is $O(n^3)$.

Another way to determine whether a negative-weight cycle exists is to create G and, without adding v_0 and its incident edges, run either of the all-pairs shortest-paths algorithms. If the resulting shortest-path distance matrix has any negative values on the diagonal, then there is a negative-weight cycle.

(b) Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

Solution: Assuming that we ran BELLMAN-FORD to solve part (a), we only need to find the vertices of a negative-weight cycle. We can do so as follows. First, relax all the edges once more. Since there is a negative-weight cycle, the d value of some vertex u will change. We just need to repeatedly follow the π values until we get back to u. In other words, we can use the recursive method given by the PRINT-PATH procedure of Section 22.2 in CLRS, but stop it when it returns to vertex u.

The running time is $O(n^3)$ to run BELLMAN-FORD, plus $O(n)$ to print the vertices of the cycle, for a total of $O(n^3)$ time.

Problem 7-2. Bicycle Tour Planning

You are in charge of planning cycling vacations for a travel agent. You have a map of n cities connected by direct bike routes. A bike route connecting cities v and u has distance $d(v, u)$. Additionally, it costs $c(v)$ to stay in city v for a single night.

A customer will provide you with the following data:

- A starting city s.
- A destination city t.
- A trip length m.
- A daily maximum biking distance $u(k)$, where $k \in [1, m]$.

Your job is to plan a tour that takes exactly m days, such that the customer does not stay in the same city two consecutive nights and does not bike more than $u(k)$ on day k. Your customer can bike through several cities on a given day, i.e. there doesn't have to be a direct route between the cities you assign on day k and day $k+1$. You also want to minimize the total cost of staying at the cities on your tour.

Give an $O(n^2(n+m))$ time algorithm to produce an m-city tour $(s = v_0, v_1 \ldots, t = v_m)$ with minimum cost $\sum c(v_i)$, such that $d(v_{i-1}, v_i) \leq u(i)$.

Solution: First, compute the all-pairs shortest path distances $\delta(i, j)$ among every pair of cities i and j. Store the values in a table.

Then construct a directed acyclic graph H with $n(m+1)$ nodes. The nodes are labeled as v_{ip}, for $i \in \{1, 2, \ldots n\}$ and $p \in \{0, 1, \ldots m\}$. Each node v_{ip} corresponds to the option of staying at city i on day p.

For any $i, j \in \{1, 2, \ldots n\}$ where $i \neq j$ and $p \in \{1, \ldots m\}$, add the edge $(v_{i(p-1)}, v_{jp})$ in the graph H if $\delta(i, j) \leq u(p)$. This means that the customer can bike from city i to city j without exceeding the daily limit. Also, for each node v_{ip}, assign a node cost c_i to the node.

The lowest cost tour is simply the path from v_{s0} to v_{tm} where the total node cost is minimum. If v_{tm} is not reachable from v_{s0}, no tour satisfying the requirements exists.

We can transform the minimum node cost path problem into a shortest path problem easily. Since the graph is directed, for every edge (u, v) we can assign the cost of node v as its edge weight. This reduces the problem to a shortest path problem, which can be computed using the shortest path algorithm on DAGs (see Section 24.2 in CLRS).

Correctness: A path on the graph from v_{s0} to v_{tm} corresponds to a bicycle tour for the m days. Specifically, each edge $(v_{i(p-1)}, v_{jp})$ along the path specifies a day trip from city i to city j on day p. Note that the edge is present in H if and only if the corresponding day trip does not violate daily mileage constraint. Also, since there is no edge between $(v_{i(p-1)}, v_{ip})$ for any i, p in the graph H, the customer will not stay at the same city on consecutive nights. The transformation of the minimum node-cost path problem into the shortest path problem preserves the cost of corresponding paths (with an offest of the cost of the source node). Therefore the shortest path algorithm on the transformed graph will compute the tour with the lowest cost.

Running Time: The all-pairs shortest paths distances can be computed in $O(n^3)$ time using the Floyd-Warshall algorithm. The graph H contains $O(nm)$ nodes and $O(n^2m)$ edges, so it takes $O(n^2m)$ time to construct the graph. Since H is a DAG, the shortest path on H can be computed in $O(n^2m)$ time. Therefore the overall running time is $O(n^3 + n^2m) = O(n^2(n + m))$.

Problem 7-3. P vs. NP

Suppose that $L_1, L_2 \in NP$ and that $L_1 <_p L_2$. For each of the following statements, determine whether it is true, false, or an open problem. Prove your answers.

(a) If $L_1 \in P$, then $L_2 \in P$.

Solution: If $P = NP$, then this is true. Otherwise, this is false: L_2 can be NP-complete. Therefore this is an open question.

(b) If $L_2 \in P$, then $L_1 \in P$.

Solution: True. Proof presented in lecture.

(c) L_2 is either NP-complete or is in P.

Solution: Open. There exist problems that are in NP, but are not known NP-complete. If $P = NP$ this is true. If $P \neq NP$ then it is false.

(d) If $L_2 <_p L_1$, then both L_1 and L_2 are NP-complete.

Solution: If $P = NP$, then this is true, since then any problem in P is NP-complete. Otherwise, it is false: we can take $L_1, L_2 \in P$, and they are not NP-complete. Therefore, this is an open problem.

(e) If L_1 and L_2 are both NP-complete, then $L_2 <_p L_1$.

Solution: True by definition of NP-completeness.

(f) Suppose there is a linear time algorithm that recognizes L_2. Then there exists a linear time algorithm to recognize L_1.

Solution: False. The reduction from L_1 to L_2, although polynomial, may take more than linear time.

Subtle Point: This does not necessarily imply that *no* linear time reduction from L_1 to L_2 exists. Essentially, this problem is equivalent to asking whether there exist any problems with superlinear lower bounds in P. That happens to be true: there do exist problems in P that cannot be solved in linear time.

This is beyond the context of 6.046. Students will get full credit for the correct answer with incomplete justification.

Problem 7-4. NP-Completeness

(a) Suppose you are given an algorithm A to solve the CLIQUE decision problem. That is, $A(G, k)$ will decide whether graph G has a clique of size k. Give an algorithm to find the vertices of a k-clique in a graph G using only calls to A, if any such k-clique exists.

Solution: Run $A(G, k)$. If A returns "false", then exit. Otherwise if A returns "true", remove an arbitrary vertex v from the graph. Run $A(G - \{v\}, k)$. If A now returns "false", then v must be in all remaining k-cliques. If A returns "true", there exists some k-clique without v. Continue this process until k vertices are found that must necessarily be in the k-clique.

(b) Prove that the following decision problem is NP-complete.

LARGEST-COMMON-SUBGRAPH: Given two graphs G_1 and G_2 and an integer k, determine whether there is a graph G with $\geq k$ edges which is a subgraph of both G_1 and G_2. (Hint: reduce from CLIQUE.)

Solution:

The problem is in NP: to prove that $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ share a subgraph of size at least k, one can exhibit a graph $H = (V^*, E^*)$ with at least k edges, and 1-1 mappings $\phi_1 : V^* \mapsto V_1$ and $\phi_2 : V^* \mapsto V_2$. Thus the certificate has size $|H| + |\phi_1| + |\phi_2|$. If n is the size of the input $|H| = O(n)$, since H is a subgraph of the given graphs, and $|\phi_i| = O(n)$ since each of them is just a list of vertices. Therefore, the size of the certificate is polynomial. To verify this certificate, check that H has at least k edges ($O(n)$ time), check that ϕ_1 and ϕ_2 are 1-1 ($O(n)$ time), and check whether for any edge $(u, v) \in E^*$, $(\phi_1(u), \phi_1(v)) \in E_1$ and $(\phi_2(u), \phi_2(v)) \in E_2$ ($O(n^2)$ time). Therefore the certificate takes polynomial time to verify.

Reduction: Given a graph G and an integer k, to determine whether G has a clique of size k, ask whether G and K_k have a common subgraph with $k(k-1)/2$ edges, where K_k is the complete graph on k nodes.

(c) Suppose you are given an algorithm B to solve the LARGEST-COMMON-SUBGRAPH decision problem. Give an algorithm to find a subgraph of size k that appears in both graphs G_1 and G_2, using only calls to B, if any such subgraph exists.

Solution: Use an approach similar to the reduction from Decisional-Clique to Search-Clique. Run $B(G_1, G_2, k)$. If B says a subgraph exists, try removing an edge from either G_1 or G_2 and re-running B. If B's output has changed, that edge must appear in all remaining subgraphs. Otherwise, it can be excluded.

Problem 7-5. Maximum Coverage Approximation

Suppose you are given a set S of size $|S| = n$, a collection $\mathcal{F}$ of m distinct subsets $\{T_1, T_2, \ldots, T_m\}$ where $T_i \subseteq S$, and a number k as input. We would like to pick k subsets from the collection that cover the maximum number of elements in S. Give a greedy, polynomial-time approximiation algorithm for this *maximum coverage problem* with ratio bound of $\min\{k, f\}$, where $f = \max_i\{|T_i|\}$. Analyze the approximation ratio achieved by your algorithm.

Solution: The following greedy algorithm achieves the $\min\{k, f\}$ ratio bound.

GREEDY-MAXIMUM-COVERAGE($S, \mathcal{F}, k$):
1 $U \leftarrow S$
2 $\mathcal{C} \leftarrow \emptyset$
3 **for** $i \leftarrow 1$ **to** k
4 select a $T_i \in \mathcal{F}$ that maximizes $|T_i \cap U|$
5 $U \leftarrow U - T_i$
6 $\mathcal{C} \leftarrow \mathcal{C} \cup \{T_i\}$
7 **return** $\mathcal{C}$

The algorithm works as follows. At each stage, the algorithm picks a subset T_i and store it in the collection $\mathcal{C}$. The set U contains, at each stage, the set of uncovered elements. The greedy approach at line 4 picks the subset that covers as many uncovered elements as possible.

The algorithm can easily be implemented in time polynomial in n, m and k. The number of iterations on lines 3-6 is k. Each iteration can be implemented in $O(n^2m)$ time. Therefore there is an implementation that runs in time $O(n^2mk)$.

The algorithm has a $\min\{k, f\}$ ratio bound. Let $\mathcal{C}$ be the solution given by the algorithm, and $\mathcal{C}^*$ be the optimal solution. If $\mathcal{C}$ covers the entire set S, then $\mathcal{C}$ is the optimal solution and we are done.

Therefore, we consider the case where $\mathcal{C}$ does not cover the entire S. At the first iteration, the algorithm will pick the largest subset with size f. Therefore the number of elements covered by $\mathcal{C}$ will be at least f. Also, at each stage, the algorithm will pick at subset that covers at least one element that has not been previously covered. Therefore, the number of elements covered by $\mathcal{C}$ will also be at least k. To sum up, the solution $\mathcal{C}$ will cover at least $\max\{k, f\}$ elements.

Now consider the optimal solution $\mathcal{C}^*$. The optimcal solution contains k subsets, and each subset contains at most f elements. Therefore the maximum number of elements covered by $\mathcal{C}^*$ is kf. It follows that the ratio bound of the greedy algorithm is:

$$\frac{kf}{\max\{k, f\}} = \min\{k, f\}.$$